CLTestCheck: Measuring Test Effectiveness for GPU Kernels

CLTestCheck: Measuring Test Effectiveness for GPU Kernels

Chao Peng and Ajitha Rajan

University of Edinburgh, UK
chao.peng@ed.ac.uk, arajan@ed.ac.uk

Abstract. Massive parallelism, and energy efficiency of GPUs, along with advances in their programmability with OpenCL and CUDA programming models have made them attractive for general-purpose computations across many application domains. Techniques for testing GPU kernels have emerged recently to aid the construction of correct GPU software. However, there exists no means of measuring quality and effectiveness of tests developed for GPU kernels. Traditional coverage criteria over CPU programs is not adequate over GPU kernels as it uses a completely different programming model and the faults encountered may be specific to the GPU architecture.
We address this need in this paper and present a framework, CLTestCheck, for assessing quality of test suites developed for OpenCL kernels. The framework has the following capabilities, 1. Measures kernel code coverage using three different coverage metrics that are inspired by faults found in real kernel code, 2. Seeds different types of faults in kernel code and measures fault finding capability of test suite, 3. Simulates different work-group schedules to check for potential deadlocks and data races with a given test suite. We conducted empirical evaluation of CLTestCheck on a collection of 82 publicly available GPU kernels and test suites. We found that CLTestCheck is capable of automatically measuring effectiveness of test suites, in terms of kernel code coverage, fault finding and revealing data races in real OpenCL kernels.

Keywords: Testing · Code Coverage · Fault Finding · Data Race · Mutation Testing · GPU · OpenCL

1 Introduction

Recent advances in the programmability of Graphics Processing Units (GPUs), accompanied by the advantages of massive parallelism and energy efficiency, have made them attractive for general-purpose computations across many application domains [18]. However, writing correct GPU programs is a challenge owing to many reasons [13]– a program may spawn millions of threads, which are clustered in multi-level hierarchies, making it difficult to analyse; programmer assumes responsibility for ensuring concurrently executing threads do not conflict by checking threads access disjoint parts of memory; complex striding patterns of memory accesses are hard to reason about; GPU work-group execution model and thread scheduling vary platform to platform and the assumptions are not explicit. As a consequence of these factors, GPU programs are difficult to analyse with existing static or dynamic approaches [13]. Static techniques are thwarted

by the complexity of the sharing patterns. Dynamic techniques are challenged by the combinatorial explosion of thread interleavings and space of possible data inputs. Given these difficulties, it becomes important to understand the extent to which a GPU program has been analysed and tested, and the code portions that may need further attention.

In this paper, we focus on GPU program testing and address concerns with respect to quality and adequacy of tests developed for GPU programs. We present a framework, CLTestCheck, that measures test effectiveness over GPU kernels written using OpenCL programming model [11]. The framework has three main capabilities. The first capability is a technique called *schedule amplification* to check execution of test inputs over several work-group schedules. Existing GPU architecture and simulators do not provide a means to control work-group schedules. The OpenCL specification provides no execution model for inter work-group interactions [20]. As a result, the ordering of work-groups when a kernel is launched is non-deterministic and there is, presently, no means for checking the effect of schedules on test execution. We provide this monitoring capability. For a test case T_i in test suite TS, instead of simply executing it once with an arbitrary schedule of work-groups, we execute it many times with a different work-group schedule in each execution. We build a simulator that can force work-groups in a kernel execution to execute in a certain order. This is done in an attempt to reveal test executions that produce different outputs for different work-group schedules which inevitably point to problems in inter work-group interactions.

The second capability of CLTestCheck is measuring code coverage for OpenCL kernels. The structures we chose to cover were motivated by OpenCL bugs found in public repositories like Github and research papers for GPU testing. We define and measure coverage over synchronisation statements, loop boundaries and branches in OpenCL kernels.

The final capability of the framework is creating mutations by seeding different classes of faults relevant to GPU kernels. We assess the effectiveness of test suites in uncovering the seeded faults.

We empirically evaluate CLTestCheck using 82 kernels and associated test input workloads from industry standard benchmarks. The schedule amplifier in CLTestCheck was able to detect deadlocks and inter work-group data races in benchmarks. We were able to detect barrier divergence and kernel code that requires further tests using the coverage measurement capabilities of CLTestCheck. Finally, the fault seeding capability was able to expose unnecessary barriers and unsafe accesses in loops.

The CLTestCheck framework aims to help developers assess how well the OpenCL kernels have been tested, kernel regions that require further testing, uncover bugs sensitive to work-group schedules. In summary, the main contributions in this paper are:

1. Schedule amplification to evaluate test executions using different work-group schedules.
2. Definition and measurement of kernel code coverage considering synchronisation statements, loop boundaries and branch conditions.
3. Fault seeder for OpenCL kernels that seeds faults from different classes. The seeded faults are used to assess the effectiveness of test suites with respect to fault finding.

4. Empirical evaluation on a collection of 82 publicly available GPU kernels, examining coverage, fault finding and inter work-group interactions.

The rest of this paper is organised as follows. We present background on the OpenCL programming model in Section 2. Related work in GPU program testing and verification is discussed in Section 3. CLTestCheck capabilities is discussed in Section 4. Experiment setup and results of our empirical evaluation is discussed in Sections 5 and 6, respectively.

2 Background

The success of GPUs in the past few years has been due to the ease of programming using the CUDA [7] and OpenCL [11] parallel programming models, which abstract away details of the architecture. In these programming models, the developer uses a C-like programming language to implement algorithms. The parallelism in those algorithms has to be exposed explicitly. We now present a brief overview of the core concepts of OpenCL, the programming model used in this paper.

OpenCL is a programming framework and standard set from Khronos, for heterogeneous parallel computing on cross-vendor and cross-platform hardware. In the OpenCL architecture, CPU-based *Host* controls multiple *Compute Devices* (for instance CPUs and GPUs are different compute devices). Each of these coarse grained compute devices consists of multiple *Compute Units* which in turn contain one or more *processing elements* (a.k.a *streaming processors*). The processing elements execute groups of individual threads, referred to as work-groups, concurrently. The functions executed by the GPU threads are called *kernels*, parameterised by thread and group id variables. OpenCL has four types of memory regions: global and constant memory shared by all threads in all work-groups, local memory shared by threads within the same work-group and private memory for each thread. Kernels cannot write to the constant memory.

GPUs have SIMT (single instruction, multiple thread) execution model that executes batches of threads (warps) in *lock-step*, i.e all threads in a work-group execute the same instruction but on different data. If the control flow of threads within the same work-group diverges, the different execution paths are scheduled sequentially until the control flows reconverge and lock-step execution resumes. Sequential scheduling caused by divergence results in a performance penalty, slowing down execution of the kernel.

Betts et al. [9] describe two specific classes of bugs that make GPU kernels harder for verification than sequential code, data races and barrier divergence. *Inter work-group data race* is referred to as a global memory location is written by one or more threads from one work-group and accessed by one or more threads from another work-group. *Intra work-group data race* is referred to as a global or local memory location is written by one thread and accessed by another from the same work-group. Barrier is a synchronisation mechanism for threads within a work-group in OpenCL and is used to prevent intra work-group data race errors. *Barrier divergence* occurs if threads in the same group reach different barriers, in which case kernel behaviour is undefined [9] and may lead to intra work-group data race.

In this paper, we focus on covering barrier functions to help detect intra work-group barrier divergence errors and revealing problems with inter work-group interactions using work-group schedule amplification.

3 Related Work

We discuss related work in the context of work-group synchronisation, verification and testing of GPU programs.

Inter Work-group Synchronisation for OpenCL Kernels. Barrier functions in the OpenCL specification [11] help synchronise threads within the same work-group. There is no mechanism, however, to synchronise threads belonging to different work-groups. One solution for this problem is to split a program into multiple kernels with the CPU executing the kernels in sequence providing implicit synchronisation. The drawback with this method is the overhead incurred in launching multiple kernels. Xiao et al. [23] proposed an implementation of inter work-group barrier that relies on information on the number of work-groups. This method is not portable as the number of launched work-groups depends on the device. Sorensen et al. [21] extended it to be portable by discovering work-group occupancy dynamically. Their implementation of inter work-group barrier synchronisation is useful when the developer knows there is interaction between work-groups that needs to be synchronised. Our contribution is in detecting undesired inter work-group interactions, not intended by the developer.

GPU Kernel Verification. Verification of GPU kernels to detect data races and barrier divergence bugs has been explored in the past. Li et al. [14] introduced a Satisfiability Modulo Theories (SMT) based approach for analysing GPU kernels and developed a tool called Prover of User GPU (PUG). The main drawback of this approach is scalability. With an increasing number of threads, the number of possible thread interleavings grows exponentially, making the analysis infeasible for large number of threads. GRace [24] and GMRace [25] were developed for CUDA programs to detect data races using both static and dynamic analysis. However, they do not support detection of inter work-group data races.

GKLEE [15] and KLEE-CL [10], based on dynamic symbolic execution, provides data race checks for CUDA and OpenCL kernels, respectively. Both tools are restricted by the need to specify a certain number of threads, and the lack of support for custom synchronisation constructs. Scalability and general applicability is a challenge with these tools.

Leung et al. [13] present a flow-based test amplification technique for verifying race freedom and determinism of CUDA kernels. For a single test input under a particular thread interleaving, they log the behaviour of the kernel and check the property. They then amplify the result of the test to hold over all the inputs that have the same values for the property integrity-inputs. The test amplification approach in [13] can check the absence of data-races, not the presence. Additionally, their approach amplifies across the space of test inputs, not work-group schedules as done in our schedule amplifier. GPUVerify [9] is a static analysis tool that transforms a parallel GPU kernel into a two-threaded predicated program with lock-step execution and checks data races over this transformed model. The drawback of GPUVerify is that it may report false alarms and has limited support for atomic operations.

Test Effectiveness Measurement. Measuring effectiveness of tests in terms of code coverage and fault finding is common for CPU programs [17, **?**]. Support for GPU programs is scarce. GKLEE is the only tool that provides support for code coverage for CUDA GPU kernels. Given a kernel, it converts it into its sequential C program version (using Perl scripts) and applies the Gcov utility supplied with GCC for measuring code coverage. This form of coverage measurement disregards the GPU programming model. In our approach, we measure coverage conforming to the OpenCL programming model. With respect to fault seeder and schedule amplification, we are not aware of any existing work that provides these capabilities for GPU kernels to help measure effectiveness of test suites. The CLTestCheck framework is discussed in the next Section 4.

4 Our Approach

In this Section, we present the CLTestCheck framework that provides capabilities for kernel code coverage measurement, mutant generation and schedule amplification. To understand the kinds of programming bugs[1] encountered by OpenCL developers, we surveyed several publicly available OpenCL kernels and associated bug fix commits. A summary of our findings is shown in Table 1. We found bugs most commonly occur in the following OpenCL code constructs: barriers, loops, branches, global memory accesses and arithmetic computations. We seek to aid the developer in assessing quality of test suites in revealing these bug types using CLTestCheck. A detailed discussion of CLTestCheck capabilities is presented in the following sections.

#	Code Structure	Bug Type	Repository
1	Barrier	Missing barriers	Winograd-OpenCL[6], histogram [13], reduction [13],OP2 [10]
2		Removing unnecessary barriers	Winograd-OpenCL[6]
3	Loop	Incorrect condition	mcxcl[4], particles[1]
4		Incorrect boundary value	clSPARSE[2]
5		Missing loop boundary	Pannotia [20]
6	Branch	Missing else branch	liboi[5]
7		Incorrect condition	mcxcl[4], ClGaussianPyramid[3]
8	Global memory access	Inter work-group data race	Parboil-spmv [16], lonestar-bfs [20], lonestar-sssp [20]
9	Arithmetic Computations	Incorrect arithmetic operators	mcxcl[4], ClGaussianPyramid[3]

Table 1: Summary of bug fixing commits we collected

4.1 Kernel Code Coverage

We define coverage over barriers, loops and branches in OpenCL code to check rigour of test suites in exercising these code structures.

[1] These are kernel bugs that violate the specification of the program or are associated with executions that lead to undefined behaviour.

Branch Coverage GPU programs are highly parallelised, executed by numerous processing elements, each of them executing groups of threads in lock step, which is very different from parallelism in CPU programs, where each thread executes different instructions with no implicit synchronisation, as seen in lock-step execution. Kernel code for all the threads is the same, however, the threads may diverge, following different branches based on the input data they process. As seen in Table 1, uncovered branches and branch conditions are an important class of OpenCL bugs. Lidbury et al. [16] report in their work that branch coverage measurement is crucial for GPU programs but is currently lacking. To address this need, we define branch coverage for GPU programs as follows,

$$branch\ coverage = \frac{\#covered\ branches}{total\ \#branches} \times\ 100\% \tag{1}$$

Branch coverage measures adequacy of a test suite by checking if each branch of each control structure in GPU code has been executed by at least one thread.

Loop Boundary Coverage In our survey of kernel bugs shown in Table 1, we found bugs related to loop boundary values and loop conditions were fairly common. For instance, bug #3 found in the `mcxcl` program allowed the loop index to access memory locations beyond the end of the array due to an erroneous loop condition. We assess adequacy of test executions with respect to loops by considering the following cases,

1. Loop body is not executed,
2. Loop body is executed exactly once,
3. Loop body is executed more than once
4. Loop boundary value is reached

$$Loop\ boundary\ coverage_{case\_i} = \frac{\#loops\ satisfying\ case\_i}{total\ \#loops} \times\ 100\% \tag{2}$$

where $case_i$ refers to one of the four loop execution cases listed above.

Barrier Coverage Barrier divergence occurs when the number of threads within a work-group executing a barrier is not the same as the total number of threads in that work-group. Kernel behaviour with barrier divergence is undefined. Barrier related bugs, missing barriers and unnecessary barriers, is a common class of GPU bugs according to our survey. We define barrier coverage as follows.

$$barrier\ coverage = \frac{\#covered\ barriers}{total\ \#barriers} \times\ 100\% \tag{3}$$

Barrier coverage measures adequacy of a test suite by checking if each barrier in GPU code is executed correctly. Correct execution of a barrier without barrier divergence, *covered barrier*, is when it is executed by *all* threads in any given work-group.

4.2 Fault Seeding

Mutation testing is known to be an effective means of estimating the fault finding effectiveness of test suites for CPU programs [12]. We generate mutations using traditional mutant operators, namely, arithmetic, relational, bitwise, logical and

assignment operator types. In Table 1, bug fixes #3, #7 and #8 show that traditional arithmetic and relational operator mutations remain applicable to GPU programs. In addition, we define three mutations specifically for OpenCL kernels: barrier mutation, image access mutation and loop boundary mutation inspired by bug fixes #1 to #5.

The barrier mutation operator we define is deletion of an existing barrier function call, to reproduce bugs similar to #1 and #2 in Table 1. OpenCL provides 2D and 3D image data structures to facilitate access to images. Multi-dimensional arrays are not supported in OpenCL. Image structures are accessed using read and write functions that take the pixel coordinates in the image as parameter. We perform image access mutations for 2D or 3D coordinates by increasing or decreasing one of the coordinates or exchanging coordinates. Finally, we define loop boundary mutations as either (1) skipping the loop, (2) allowing n-1 iterations of the loop and (3) allowing n+1 iterations of the loop where n is the number of iterations when the loop boundary is reached. The mutant operators we use in this paper are summarised in Table 2

Type of Operator		Mutants
Arithmetic	Binary	+, -, *, /, %
	Unary	-(negation), ++, --
Relational		<, >, ==, <=, >=, !=
Logical		&&, ||, !
Bitwise		&, |, ^, ~, <<, >>
Assignment		=, +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=
Barrier		Delete barrier function call
Image coordinates		Change coordinates when accessing images
Loop boundary		Change the boundary value in loop condition check

Table 2: Summary of mutation operators

4.3 Schedule Amplification

When a kernel execution is launched the GPU schedules work-groups on compute units in a certain order. Presently, there is no provision for determining this schedule or setting it in advance. The scheduler makes the decision on the fly subject to availability of compute units and readiness of work-groups for execution. The order in which work-groups are executed with the same test input can differ every time the kernel is executed. OpenCL specification has no execution model for inter work-group interactions and provides no guarantees on how work-groups are mapped to compute units. In our approach, we execute each test input over a set of schedules. In each schedule, we fix the work-group that should execute first. All other work-groups wait till it has finished execution. The work-group going first is picked so that we achieve a uniform distribution over the entire range of work-groups in the set of schedules. The order of execution for the remaining work-groups is left to the scheduler. For a test case, T over a kernel with G work-groups, we will generate N schedules, with $N < G$, such that a different work-group is executed first in each of the N schedules. The number of schedules, N, we generate is much lesser than the total number of schedules which is typically infeasible to check. The reason we only fix

the first work-group in the schedule is because, most data races or deadlocks involve interactions between two work-groups. Fixing one of them and picking a different work-group each time, significantly reduces the search space of possible schedules. We cannot provide guarantees with this approach. However, with little extra cost we are able to check significantly more number of schedules than is currently possible. We believe this approach will be effective in revealing issues, if any, in inter work-group interactions.

To illustrate this, we consider a kernel co running on four work-groups. The CLTestCheck schedule amplifier will insert code on the host and GPU side, shown in Listings 1.1 and 1.2, to generate different work-group schedules.

Listing 1.1: Schedule OpenCL kernel (CPU-side)

```
// Generate a value in the range of [0,4)
int target_group = randint(4);
// Pass the value as a macro to GPU code
sprintf(clOptions,"-DTARGET_GROUP=%d", target_group);
```

Listing 1.2: Schedule OpenCL kernel (GPU-side)

```
if (my_group_id == TARGET_GROUP){
  // Original code here executed by target group
  A[(1 - buf) * 4 + tid] = A[buf * 4 + (tid + 1) % 4];
  atom_increase(num_threads_finishes);
} else {
  while (num_threads_finishes != group_size) continue;
  // Original code executed by other groups
  A[(1 - buf) * 4 + tid] = A[buf * 4 + (tid + 1) % 4];
}
```

In this example, before the GPU kernel is launched, the host side generates a random value in the range of available work-group ids. This value is the id of the selected work-group to be executed first and is passed to the kernel code using a macro definition. On the kernel side, each thread determines if it belongs to the selected work-group. Threads in the selected work-group proceed with executing the kernel code while threads belonging to other work-groups wait. After the selected work-group completes execution, the remaining work-groups execute the original kernel in an order based on mapping to available compute units (occupancy bound execution model [21]). With different work-group schedules generated by the schedule amplifier, we were able to detect the presence of *inter* work-group data races using a *single* GPU platform. Betts et al. [9], on the other hand, focus on intra work-group data races on different GPU platforms.

4.4 Implementation

CLTestCheck is implemented using Clang LibTooling [8]. We instrument OpenCL kernel source code to measure coverage, generate mutations and multiple work-group schedules automatically.

Coverage Measurement. To record branches, loops and barriers executed within each kernel when running tests, we instrument the kernel code with data

structures and statements recording the execution of these code structures. For each work-group, we introduce three local arrays, whose size is determined by the number of branches, loops and barriers accessible by threads in that work-group. To measure branch coverage, we add statements at the beginning of each then- and else-branch to record whether that branch is enabled. Similarly, statements to record the number of iterations of loops are added at the beginning of each loop body. At the end of the kernel, the information contained in the data structures is processed to compute coverage

Fault Seeder and Mutant Execution. The CLTestCheck fault seeder generates mutants and executes them with each of the tests in the test suite to compute mutation score, as the fraction of mutants killed. The CLTestCheck fault seeder translates the target kernel source code into an intermediate form where all the applicable operators are replaced by a template string containing the original operator, its ID and type. The tool then generates mutants from this intermediate form. Once mutants are generated, the tool executes each of the mutant files and checks if the test suite kills the mutant. We term the mutant as killed if one of the following occurs: program crashes, deadlocks or produces a result different from the original kernel code.

Schedule Amplification. As mentioned earlier, we generate several schedules for each test execution by requiring a target work-group to execute the kernel code first and then allowing other work-groups to proceed. The target work-group is selected uniformly across the input space of work-group ids. To achieve coverage of this input space, we partition work-group ids into sets of 10 work-groups. Thus if we have N work-groups, we partition them into $N/10$ sets. The first set has work-group ids 0 to 9, the second set has ids 10 to 19 and so on. We then randomly pick a target work-group, W_t, from each of these sets to go first and generate a corresponding schedule of work-groups, $\{W_t, S_{N-1}\}$, where S_{N-1} refers to the schedule of remaining $N-1$ work-groups generated by the GPU execution model which is non-deterministic. For $N/10$ sets of work-groups, we will have $N/10$ schedules of the form $\{W_t, S_{N-1}\}$ (a W_t first schedule). The test input is executed using each of these $N/10$ W_t first schedules. Due to the non-deterministic nature of S_{N-1}, we repeat the test execution with a chosen W_t first schedule 20 times. This will enable us to check if the execution model generates different S_{N-1} and evaluate executions with 20 such orderings.

5 Experiment

In our experiment, we evaluate the feasibility and effectiveness of the coverage metrics, fault seeder and work-group schedule amplifier proposed in Section 4 using OpenCL kernels from industry standard benchmark families and their associated test suites. We investigate the following questions:

Q1. Coverage Achieved: *What is the branch, barrier and loop coverage achieved by test suites over OpenCL kernels in our subject benchmarks?*
To answer this question, we use our implementation to instrument and analyse kernel source code to record visited branches, barrier functions, loop iterations along with information on executing work-group and threads.

Q2. Fault Finding: *What is the mutation score of test suites associated with the subject programs?*

For each benchmark, we generate all possible mutants by analysing the kernel source code and applying the mutation operators, discussed in Section 4, to eligible locations. We then assess number of mutants killed by the tests associated with each benchmark. To check if a mutant is killed, we compared execution results between the original program and mutant.

Q3. Deadlocks and Data Races: *Can the tests in the test suite give rise to unusual behaviour in the form of deadlocks or data races?* Deadlocks occur when two or more work-groups are waiting on each other for a resource. Inter work-group data races occur when test executions produce different outputs for different work-group schedules. For each test execution in each benchmark, we generate $20 * N/10$ different work-group schedules, where N is total number of work-groups for the kernel, and check if the outputs from the execution change based on work-group schedule.

Subject Programs. We used the following benchmarks for our experiments, 1. Nine scientific benchmarks with 23 OpenCL kernels from Parboil benchmark suite [22], 2. `scan` benchmark [19], with 3 kernels, that computes parallel prefix sum, 3. Five applications containing 13 kernels from Rodinia benchmark suite for heterogeneous computing, 4. 20 benchmarks from PolyBench with 43 kernels spanning linear algebra, data mining and stencil computations.

We ran our experiments on Intel CPU (i5-6500) and GPU (HD Graphics 530) using OpenCL SDK 2.0.

6 Results and Analysis

For each of the subject programs presented in Section 5, we ran the associated test suites and report results in terms of coverage achieved, fault finding and overhead incurred with CLTestCheck framework. We executed the test suites 20 times for each measurement. Our results in the context of the questions in Section 5 is presented below.

6.1 Coverage Achieved

Branch and Loop coverage (with 0, exactly 1 and > 1 iterations) for each of the subject programs in the three benchmark suites[2] is shown in the plots in Figure 1. The first row shows branch coverage, the second loop coverage. Mutation score and surviving mutation types shown in the last two rows of Figure 1 is discussed in the next Section 6.2.

Barrier Coverage is not shown in the plots since for all, except one, applications with barriers, the associated test suites achieved 100% barrier coverage. The only subject program with less than 100% barrier coverage was `scan`, which had 87.5% barrier coverage. The uncovered barrier is in a loop whose condition does not allow some threads to enter the loop, resulting in barrier divergence between threads. We find that less than 100% barrier coverage is a useful indicator of barrier divergence in code.

Branch Coverage. For most subject programs in Parboil and Scan/Rodinia, test suites achieve high branch coverage ($> 83\%$). The `histo` benchmark is an

[2] 20 applications in Parboil counting different test suites separately, 6 in Scan/Rodinia, and 20 in PolyBench

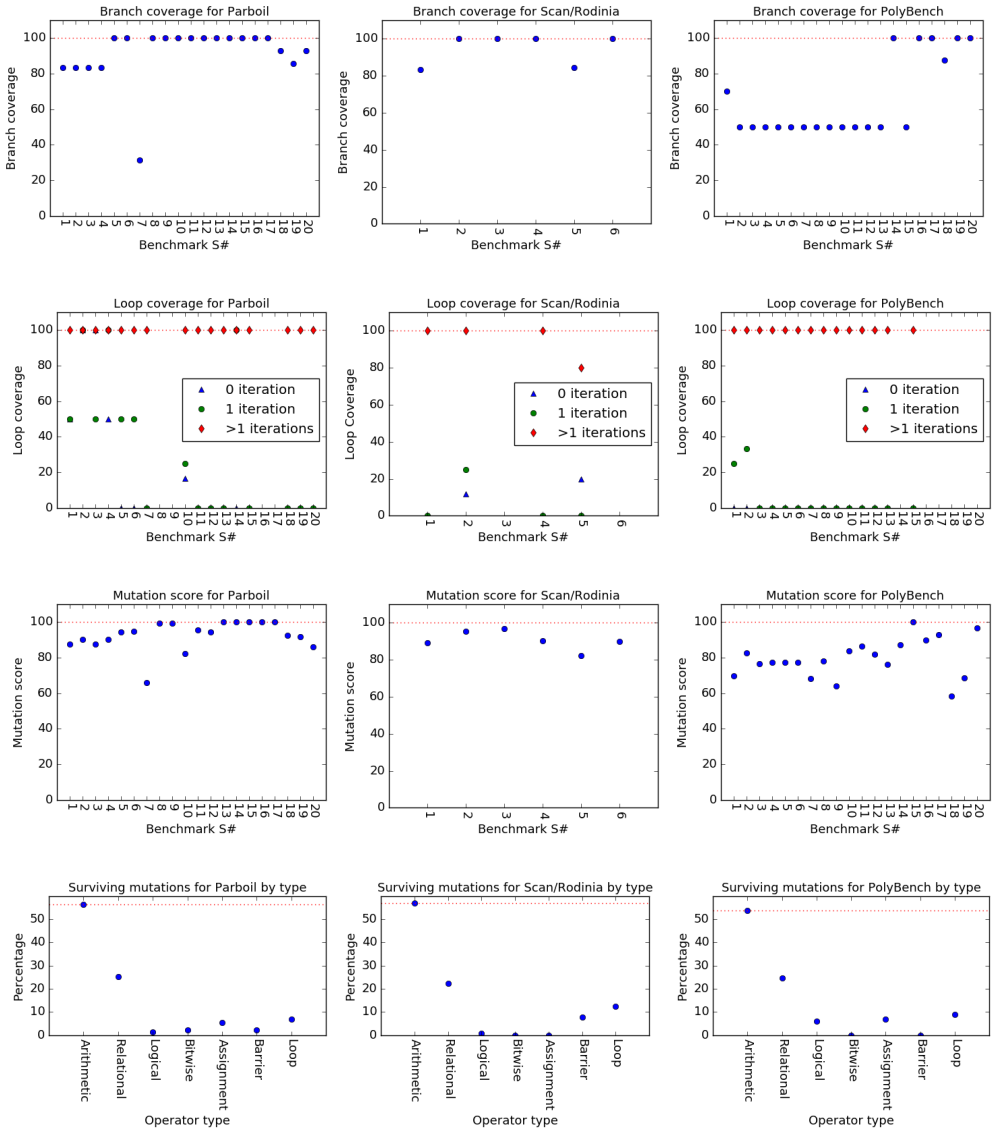

Fig. 1: Coverage achieved - Branch and Loop, mutation score and percentage of surviving mutations by type for each subject program in the 3 benchmark suites.

outlier with a low branch coverage of 31.6%. Its kernel function, `histo_main`, contains 20 branches in a code block handling an exception condition (overflow). The test suite provided with `histo` does not raise the overflow exception, and as a result, these branches are never executed. We found uncovered branches in other applications, with $> 80\%$ coverage, in Parboil and Scan/Rodinia to also result from exception handing code that is not exercised by the associated test data.

Branch coverage achieved for 13 of the 20 applications in PolyBench is at 50%. This is very low compared with other benchmark suites. Upon investigating the kernel code, we found that all the uncovered branches reside within a condition check for out of range array index. Tests associated with a majority of the applications did not check out of range array index access, resulting in low branch coverage.

Loop Coverage. Test suites for nearly all applications (with loops) execute loops more than once. Thus, coverage for > 1 iterations is 100% for all but one of the applications, `srad` in Rodinia suite, that has 80%. The uncovered loop in `srad` is in an uncovered then-branch that checks exception conditions. We also checked if the boundary value in loop conditions is reached when > 1 iterations is covered by test executions. We found `pathfinder` in Rodinia to be the only application to have full coverage for > 1 iterations but not reach the boundary value. The unusual scenario in `pathfinder` is because one of the loops is exited using a break statement.

We find that test suites for most applications are unable to achieve any loop coverage for 0 and exactly 1 iteration. The boundary condition for most loops is based on the size of the work-groups which is typically much greater than 1. As a result, test suites have been unable skip the loop or execute it exactly once. The only exceptions were applications in the Parboil suite - `bfs, cutcp, mri-gridding, spmv`, and two applications in Rodinia- `lud, srad`, that have boundary values dependent on variables that maybe set to 0 or 1.

Overhead. For each benchmark and associated test suite, we assessed overhead introduced by our approach. We compared time needed for executing the benchmark with instrumentation and additional data structures that we introduced for coverage measurement against the original unchanged benchmark. Overhead varied greatly across benchmarks and test suites. Overhead for Parboil and Rodinia benchmarks was in the range of 2% to 118%. Overhead was lower for benchmarks that took longer to execute as the additional execution time from instrumentation is a smaller fraction of the overall time. Overhead for most programs in PolyBench ranges from 2% to 70%, which is similar to Parboil and Rodinia benchmarks. The overhead for `lu`, `fdtd-2d` and `jacobi-2d-imper` programs are $> 100\%$. The code for kernel computations in these benchmarks is small with fast execution. Consequently, the relative increase in code size and execution time after instrumentation with CLTestCheck is high.

6.2 Fault Finding

Fault finding for the subject programs is assessed using the mutants we generate with the fault seeder, described in Section 4. The mutation score, percentage of mutants killed, is used to estimate fault finding capability of test suites associated with the subject programs. Each test suite associated with a benchmark is run

20 times to determine the killed mutants. A mutant is considered killed if the test suite generates different outputs on the mutant than the original program in *all* 20 repeated runs of the test suite. In addition to killed mutants, we also report results on "Undecided Mutants", that refers to mutants that are killed in at least one of the executions of the test suite, but *not all* 20 repeated executions. Changes in GPU thread scheduling between runs causes this uncertainty. We do not count the undecided mutants towards killed mutants in the mutation score. Mutation score for all subject programs in each benchmark suite is shown in the third row of plots in Figure 1.

Mutation Score. In general, we find that test suites for subject programs achieving high branch, barrier and loop coverage also have high mutation score. For instance, for `spmv` and `stencil`, their test suites achieving 100% coverage, also achieved 100% mutation score. An instance of a program that does not follow this trend is `mri-gridding` that has 100% branch, barrier, and loop (> 1 iterations) coverage but only 82% mutation score. On analysing the survived mutants, we found a significant fraction (160 out of 232) were arithmetic operator mutations within a function named *kernel_value* that contained variables defining a fourteenth-order polynomial and a cubic polynomial. Effect of mutations on the polynomials did not propagate to the output of the benchmark with the given test suite. The `histo` program with low branch coverage, 100% barrier and loop coverage has 65.9% mutation score. Nearly two thirds of the branches in `histo` cannot be reached by the input data, as a result, all the mutations in the untouched branches is not killed, resulting in a low mutation score. A few of the programs in PolyBench have mutation scores that are between 60 – 70%. In these programs, most surviving mutations are arithmetic operator mutations.

As seen in the last row of Figure 1 showing surviving mutations by operator type, arithmetic operators are the dominant surviving mutations in all three benchmark suites. Control flow adequate tests can kill arithmetic operator mutations only if they propagate to a control condition or the output. Data flow coverage may be better suited for estimating these mutations. Around 20% of relational operator mutations also survive in our evaluation. Most of the surviving relational operator mutations made slight changes to operators, such as $<$ to $<=$, or $>$ to $>=$ and vice versa. The test suites provided with the benchmarks missed such boundary mutations.

Undecided mutants occur during executions of 9, out of the 46 subject programs and test suites across all three benchmark suites. Number of undecided mutants during the 9 executions is generally small ($<= 5$). The only exception is `tpacf` in the Parboil benchmark suite, that resulted in 18 undecided mutants when executing one of its test suite. Undecided mutants point to non-deterministic behaviour in the kernel, that is dependent on GPU thread execution model. A large number of undecided mutants is alarming and developers should examine kernel code more closely to ensure that the behaviour observed is as intended.

Barriers were not used in all benchmarks. Only 5 out of the 9 benchmarks in Parboil, and 4 of the 6 in Scan/Rodinia had barriers. PolyBench programs did not use any barriers. Mutations removed barrier function calls in these benchmarks and we ecorded the number of mutants killed by test suites. Percentage of killed barrier mutations is generally low across all benchmarks with barriers. For instance, removing 2 out of 3 barriers in the `histo` program in Parboil,

and removing all barriers in the `cutcp` program had no effect on outputs of the respective program executions. This may either mean that the test suites are inadequate with respect to the barrier mutations or it could be an indication that these barriers are superfluous with respect to program outputs, and the need for synchronisation should be further justified. For the programs in our experiment, we found barriers, whose mutations survived, to be unecessary.

Coverage versus Mutation Score. The plots in Figure 1 illustrate total mutation score over all types of mutations for each subject program and test suite. We also compute mutation scores specifically for branches, barriers, and loops using mutations relevant to them. We do this to compare against branch, barrier and loop coverage achieved for each of the subject programs. We found that mutation score for branches closely follows branch coverage for most subject programs. Outliers include `adi`, `nn`, `convolution-2d` and `convolution-3d`. Mutations that change $<$ to $<=$ are not killed in these kernels; these comprise one third of all branch mutations.

Mutation score for barriers is quite different from barrier coverage. This is because test suites are able to execute the barriers and achieve coverage. However, they are unable to produce different outputs when the barriers are removed. This may be a problem with the superfluous manner in which barriers are used in these programs.

Loop coverage with > 1 iterations is 100% for all but one subject program (`srad` in Rodinia). Mutation score for loops on the other hand is variable. In general, tests achieving loop coverage are unable to reveal loop boundary mutations. Histo and srad are worth noting with high loop coverage but low loop mutation scores. We find that mutations to the loop boundary value in these two benchmarks survive, which implies that access to loop indices outside the boundary go unchecked in these programs. These unsafe values of loop indices should be disallowed in these kernels and loop boundary mutations in our fault seeder help reveal them.

6.3 Schedule Amplification: Deadlocks and Data Races

Kernel Deadlocks: When we used the CLTestCheck schedule amplifier on our benchmarks, we found kernel executions deadlock when the work-group ID selected to go first exceeds the number of available compute units. As there are no guarantees on how work-groups are mapped to compute units, we allow work-group IDs exceeding number of compute units to go first in some test executions using our schedule amplifier. However, it appears that the GPU makes unstated assumptions on what work-group IDs are allowed to go first. As noted by Sorenson et al. [21], "execution of large number of work-groups is in any *occupancy bound* fashion, by delaying the scheduling of some work-groups until others have executed to completion." They observed deadlocks in kernel execution due to inter work-group barriers. However, in the benchmarks in our evaluation, there is no explicit inter work-group barrier. It may be the case that developers made implicit assumptions on inter work-group barriers using the occupancy bound model and our schedule amplification approach violates this assumption. Nevertheless, our finding exposes the need for an inter work-group execution model that explicitly states the details and assumptions related to mapping of work-groups to compute units for a given kernel on a given GPU platform.

Inter Work-group Data Races: We were able to reveal a data race in the `spmv` application from the Parboil benchmark suite. We found that when work-groups 0 or 1 are chosen to go first in our schedules, the kernels execution always produces the same result. However, when we pick other work-group ids to go first, the test output is not consistent. Among twenty executions for each schedule, the frequency of producing correct output varies from 45% to 70%.

We observe similar behaviour in the `tpacf` application in Parboil when we delete the last barrier function call in the kernel. The kernel execution produces consistent outputs when we pick work-group 0 or 1 to go first. When we pick other work-groups to go first using our schedule amplifier, the kernel execution results are non-deterministic.

We observe no unusual behaviour in any of the PolyBench programs. These programs split the computation into multiple kernels and the CPU program launches GPU kernels one by one. The transfer of control from the GPU to the CPU between kernels acts like a barrier as the CPU will wait until a kernel finishes before launching the next kernel. In addition, care has been taken in the kernel code to ensure threads do not access the same memory location. As a result, we observe no data races in PolyBench with our schedule amplifier.

7 Conclusion

We have presented the CLTestCheck framework for measuring test effectiveness over OpenCL kernels with capabilities to measure code coverage, fault seeding and mutation score measurement, and finally amplify the execution of a test input with multiple work-group schedules to check inter work-group interactions. Our empirical evaluation of CLTestCheck capabilities with 82 publicly available kernels revealed the following,

1. The schedule amplifier was able to detect deadlocks and inter work-group data races in Parboil benchmarks when higher work-group ids were forced to execute first. This finding emphasizes the need for transparency and clearly stated assumptions on how work-groups are mapped to compute units.
2. Barrier coverage served as a useful measure in identifying barrier divergence in benchmarks (`scan`).
3. Branch coverage pointed to inadequacies in existing test suites and found test inputs for exercising error handling code were missing.
4. Across all benchmark suites, we found arithmetic operator and relational operator mutations that changed $<$ to $<=$, $>$ to $>=$ or vice versa were hard to kill. More rigorous test suites to handle these mutations are needed.
5. The use of barrier mutations revealed several instances of unnecessary barrier use. Barrier usage and its implications is not well understood by developers. Barrier mutations can help reveal incorrect barrier uses.
6. Loop boundary mutations helped reveal unsafe accesses to loop indices outside the loop boundary.

In sum, the CLTestCheck framework is an automated, effective and useful tool that will help developers assess how well OpenCL kernels have been tested, kernel regions that require further testing, uncover bugs with respect to work-group schedules. In the future, we plan to add further metrics, like data flow coverage with work-group schedule, to strengthen test adequacy measurement.

References

1. Cinematic particle effects with opencl, https://github.com/hortont424/particles
2. clsparse: a software library containing sparse functions written in opencl, https://github.com/clMathLibraries/clSPARSE
3. Experiments on gaussian pyramid implemented using opencl, https://github.com/twitwi/ClGaussianPyramid
4. Monte carlo extreme for opencl (mcxcl), https://github.com/fangq/mcxcl
5. Opencl interferometry library (liboi), https://github.com/bkloppenborg/liboi
6. Winograd-based convolution implementation in opencl, https://github.com/csehydrogen/Winograd-OpenCL
7. Cuda zone (Sep 2017), https://developer.nvidia.com/cuda-zone
8. clang: a c language family frontend for llvm (March 2018), http://clang.llvm.org/
9. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: Gpuverify: a verifier for gpu kernels. In: ACM SIGPLAN Notices. vol. 47, pp. 113–132. ACM (2012)
10. Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic testing of opencl code. In: Haifa Verification Conference. pp. 203–218. Springer (2011)
11. Group, K.O.W.: The opencl specification version 2.2 (May 2017)
12. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE transactions on software engineering **37**(5), 649–678 (2011)
13. Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., Lerner, S.: Verifying gpu kernels by test amplification. In: ACM SIGPLAN Notices. vol. 47, pp. 383–394. ACM (2012)
14. Li, G., Gopalakrishnan, G.: Scalable smt-based verification of gpu kernel functions. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. pp. 187–196. ACM (2010)
15. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: Gklee: concolic verification and test generation for gpus. In: ACM SIGPLAN Notices. vol. 47, pp. 215–224. ACM (2012)
16. Lidbury, C., Lascu, A., Chong, N., Donaldson, A.F.: Many-core compiler fuzzing. ACM SIGPLAN Notices **50**(6), 65–76 (2015)
17. Rajan, A., Heimdahl, M.P.: Coverage metrics for requirements-based testing. University of Minnesota (2009)
18. Rajan, A., Sharma, S., Schrammel, P., Kroening, D.: Accelerated test execution using gpus. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. pp. 97–102. ACM (2014)
19. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for gpu computing. In: Graphics hardware. vol. 2007, pp. 97–106 (2007)
20. Sorensen, T., Donaldson, A.F.: The hitchhiker's guide to cross-platform opencl application development. In: Proceedings of the 4th International Workshop on OpenCL. p. 2. ACM (2016)
21. Sorensen, T., Donaldson, A.F., Batty, M., Gopalakrishnan, G., Rakamarić, Z.: Portable inter-workgroup barrier synchronisation for gpus. In: ACM SIGPLAN Notices. vol. 51, pp. 39–58. ACM (2016)
22. Stratton, J.A., Rodrigues, C., Sung, I.J., Obeid, N., Chang, L.W., Anssari, N., Liu, G.D., Hwu, W.m.W.: Parboil: A revised benchmark suite for scientific and commercial throughput computing. Center for Reliable and High-Performance Computing **127** (2012)
23. Xiao, S., Feng, W.c.: Inter-block gpu communication via fast barrier synchronization. In: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on. pp. 1–12. IEEE (2010)

24. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: Grace: a low-overhead mechanism for detecting data races in gpu programs. In: ACM SIGPLAN Notices. vol. 46, pp. 135–146. ACM (2011)
25. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: Gmrace: Detecting data races in gpu programs via a low-overhead scheme. IEEE Transactions on Parallel and Distributed Systems **25**(1), 104–115 (2014)